Handling Bit-Propagating Operations in Bit-Vector Reasoning

Alexander Nadel

Intel Corporation, P.O. Box 1659, Haifa 31015, Israel
alexander.nadel@intel.com

Abstract

Our aim is to improve bit-vector reasoning in modern SMT solvers. We enhance bit-vector preprocessing by introducing algorithms that explicitly handle an important class of bit-vector operations which we call *bit-propagating*. Such operations fulfill the following property: each output bit is either a bit of one of the inputs or a constant (0 or 1). We identified ten bit-propagating operations in the SMT-LIB 2.0 language; these operations are encountered frequently in practice. Our algorithms seek to improve the run-time of SMT solvers by simplifying the problem that is eventually provided to the underlying SAT solver. Empirical evaluation of our algorithms reveals a performance boost across a variety of SMT-LIB benchmark families.

1 Introduction

Bit-vector (abbr., BV) reasoning is widely used in practice. Over 48% out of more than 93,000 benchmarks in SMT-LIB [4] are either plain BV benchmarks or combine the BV theory with the theory of arrays (where 33% are plain BV benchmarks) [1]. Bit-vector reasoning is supported by a variety of solvers, such as Boolector [7], STP [12], Mathsat [8] and others. In the eager approach to BV solving (used by Boolector and STP, for example), the solver preprocesses the word-level formula, then translates the simplified formula to Conjunctive Normal Form (CNF) and solves it with a SAT solver. In this paper we identify an important class of BV operations and propose an efficient way of handling them in the preprocessor. We restrict further discussion in this paper to eager SMT solvers applied to solving bit-vector benchmarks that conform to the QF_BV logic syntax [2]. However, our results are applicable whenever BV reasoning is required.

The central notion of our paper is that of a *bit-propagating* operation. A bit-propagating operation fulfills the following property: each output bit is either a bit of one of the inputs or a constant 0 or 1 (a more precise definition appears in Section 3.1). We identified 10 bit-propagating operations amongst the 38 operations over bit-vectors supported in QF_BV (that is, the union of the 35 operations in the Fixed_Size_BitVectors theory and the 3 operations in the Core theory). The two basic bit-propagating operations are `concat` and `extract`, while the others comprise two rotation operations, `repeat`, and three shift operations (we consider the shift operations to be bit-propagating only when the shift is by a constant). A full list of bit-propagating operations is provided in Fig. 1. We found that bit-propagating operations appear frequently in practice. See Table 1 for more details.

As an example of a bit-propagating operation, consider the shift left operation `bvshl`. Assume that a bit-vector variable of width 4 $v = \left[v^{[3]}, v^{[2]}, v^{[1]}, v^{[0]}\right]$ ($v^{[0]}$ is the least significant bit) is shifted by the constant 3. The result would be $v = \left[v^{[0]}, 0, 0, 0\right]$. Clearly, our property holds: bits 0 through 2 of the output are constants, while bit 3 of the output is bit 0 of the input.

We assume that the SMT solver maintains a directed acyclic graph to represent the input formula, where each node correponds to either a bit-vector constant, an input bit-vector variable, or an internal bit-vector variable created as a result of applying an operation. We call a variable *bit-propagating* if it was created as a result of applying a bit-propagating operation.

The main observation behind our work is that given a nested application of bit-propagating operations, the bits of the resulting variable can always be expressed in terms of bits of non-bit-propagating variables and the constants 0 and 1. For example, assume that the following variables were created in the order specified, with the width being 4 bits for every variable except v_3:

1. $v_1 := \texttt{bvadd}(u_1, u_2)$ (bit-vector addition of two previously declared variables)
2. $v_2 := \texttt{bvshl}(v_1, 1)$ (a shift by a constant)
3. $v_3 := \texttt{extract}(v_2, 3, 2)$ (extracting bits 2 through 3)

Both v_2 and v_3 can be expressed in terms of the bits of v_1 and the constant 0. More specifically, we have $v_2 = \left[v_1^{[2]}, v_1^{[1]}, v_1^{[0]}, 0\right]$ and $v_3 = \left[v_1^{[2]}, v_1^{[1]}\right]$.

Our main idea is of associating each newly created variable with the so-called *Bit-Propagating Normal Form (BPNF)* which expresses the variable in terms of bits of non-bit-propagating variables and constants. BPNFs of all the variables are stored in a hash table. We propose a succint representation for BPNF in Section 3.1.

The idea of associating a normal form with bit-vector expressions over `concat` and `extract` operations is well known [9, 6, 5]. In particular, the concatenation normal form [6], detailed in [11], is largely similar to our BPNF. The added value of our proposal is that it extends the normal form to variables created with eight additional operations available in the modern SMT-LIB 2.0 language and integrates the normal-form-based reasoning into a modern SMT solver.

One advantage of maintaining a BPNF is that one can avoid creating new variables when bit-vector variables with identical BPNFs are created through different sequences of bit-propagating operations. Let us continue our example:

4. $v_4 := \texttt{repeat}(v_1, 2)$
5. $v_5 := \texttt{extract}(v_4, 6, 5)$

We have $v_4 = \left[v_1^{[3]}, v_1^{[2]}, v_1^{[1]}, v_1^{[0]}, v_1^{[3]}, v_1^{[2]}, v_1^{[1]}, v_1^{[0]}\right]$ and $v_5 = \left[v_1^{[2]}, v_1^{[1]}\right]$. Hence v_5 is identical to v_3. Imagine that the SMT solver is required to create an internal variable coresponding to the operation $\texttt{extract}(v_4, 6, 5)$. After calculating the BPNF and looking in the BPNF hash table, the solver can conclude that a new internal variable is not required, since v_3 can be used instead.

Another advantage of maintaining a BPNF is that when the formula is translated to CNF, we create new CNF variables only for non-bit-propagating word-level variables, since we can express all bits of the bit-propagating variables in terms of constants and bits of non-bit-propagating variables. Hence, maintaining BPNFs is expected to reduce the number of CNF variables and clauses, thus simplifying the problem for the SAT solver.

An alternative way of refraining from creating new CNF variables for the outputs of bit-propagating operations would simply be to reuse CNF variables during the bit-blasting stage. In our example, in order to represent v_3 in CNF, one could use the CNF variables created

to represent $v_1^{[2]}$ and $v_1^{[1]}$.[1] However, such an approach would not have the first advantage of maintaining a normal form we mentioned, that is, creating only one actual word-level variable for variables with identical BPNFs created through different sequences of bit-propagating operations. Our proposal has this advantage over any approach based on hashing individual bits.

We implemented our algorithms in Intel's new eager BV solver Hazel, whose architecture is largely similar to that of Boolector and STP, and tested their usefulness on benchmark families from SMT-LIB [3]. We show that applying our algorithms results in a performance boost across a variety of SMT-LIB families. We also show that on these families, Hazel is usually faster than the state-of-the-art academic solvers.

In what follows, Section 2 contains preliminaries. Section 3 describes the core algorithms we propose. Experimental results are provided in Section 4. Section 5 concludes our paper.

2 Preliminaries

We need to define notions related to the basics of BV reasoning.

Definition 1 (Bit, Bit-Vector Variable, Constant)**.** *A* bit *is a Boolean variable (it can be interpreted as 0 or 1). A* bit-vector variable v *of width* $|v|$ *is a sequence* $v = \left\{v^{[|v|-1]}, \ldots, v^{[1]}, v^{[0]}\right\}$*, where* $v^{[i]}$ *is a bit for each* $|v| > i \geq 0$*. The set of all bit-vectors is denoted by* $\mathcal{B}$*. A* constant *is a bit-vector variable, whose every bit is interpreted as 0 or 1. The set of all constants is denoted by* $\mathcal{C}$*.*

We will sometimes refer to bit-vector variables as either bit-vectors or variables. We consider bits to be bit-vectors of width 1. We denote by 0^w a constant of width w whose every bit is 0.

It is not hard to check that the domain of every operation supported in QF_BV is a cross product of one, two or three bit-vectors (we consider variables of sort Bool to be bit-vectors of width 1) and zero, one or two natural numbers, while the range comprises a bit-vector. We denote by $\mathcal{N}$ the set of natural numbers (including 0), and by $bits(v)$ the set of all bits of a bit-vector v. Formal definitions of a bit-propagating operation and a bit-range follow.

Definition 2 (Bit-Propagating Operation)**.** *An operation* $\omega : \underbrace{\mathcal{B} \times \mathcal{B} \times \ldots \mathcal{B}}_{1 \leq k \leq 3} \times \underbrace{\mathcal{N} \times \mathcal{N} \times \ldots \times \mathcal{N}}_{0 \leq l \leq 2} \rightarrow \mathcal{B}$ *is* bit-propagating *if for every application of* ω*:* $\omega(v_k \in \mathcal{B}, \ldots, v_1 \in \mathcal{B}, x_l \in \mathcal{N}, \ldots, x_1 \in \mathcal{N}) = u \in \mathcal{B}$*, for every* $|u| > i \geq 0$ *it holds that* $u^{[i]} \in \{0,1\} \cup bits(v_k) \cup \ldots \cup bits(v_2) \cup bits(v_1)$ *and that* $u^{[i]}$ *can be computed at the time the operation is applied.*

Definition 3 (Bit-Range)**.** *Let* v *be a bit-vector of width* n*. Then for every* $n > i \geq 0$ *and* $n > j \geq i$*, the sequence* $v^{[j:i]} = \left\{v^{[j]}, \ldots, v^{[i+1]}, v^{[i]}\right\}$ *is a* bit-range *of* v*.*

Fig. 1 provides the set of all bit-propagating operations in the SMT-LIB 2.0 language. The shift operations in the language (`bvshl`–shift left, `bvlshr`–logical shift right, and `bvashr`–arithmetic shift right) support shifting by an arbitrary bit-vector. However, we consider shifts to be bit-propagating only when the shift is by a constant, since otherwise the match between the output and the input bits is not known at the time the operation is applied. Hence, in Fig. 1, the shift operations receive a natural number as their second parameter. Note that all the operations can be expressed in terms of one or more applications of `extract` and/or

[1] Unfortunately, we are not aware of any publications containing details of bit-blasting algorithms applied by SMT solvers. To the best of our knowledge, some SMT solvers re-use CNF variables while handling the `extract` and `concat` operations during bit-blasting, but not the other eight bit-propagating operations we identified.

1. $\texttt{concat}(v_1 \in \mathcal{B}, v_2 \in \mathcal{B}) = \left\{v_1^{[|v_1|-1]}, \ldots, v_1^{[1]}, v_1^{[0]}, v_2^{[|v_2|-1]}, \ldots, v_2^{[1]}, v_2^{[0]}\right\}$
2. $\texttt{extract}(v \in \mathcal{B}, m \in \mathcal{N}, l \in \mathcal{N}) = v^{[m:l]}$, where $|v| > m, l \geq 0$ and $m \geq l$
3. $\texttt{repeat}(v \in B, s \in \mathcal{N}) = \texttt{concat}(\texttt{repeat}(v, s-1), v)$ for $s > 1$; $\texttt{repeat}(v \in B, 1) = v$; $\texttt{repeat}(v, 0)$ is undefined
4. $\texttt{zero\_extend}(v \in \mathcal{B}, s \in \mathcal{N}) = \texttt{concat}(0^s, v)$ for $s > 0$; $\texttt{zero\_extend}(v, 0) = v$
5. $\texttt{sign\_extend}(v \in \mathcal{B}, s \in \mathcal{N}) = \texttt{concat}(\texttt{repeat}(v^{[|v|-1]}, s), v)$ for $s > 0$; $\texttt{sign\_extend}(v, 0) = v$
6. $\texttt{bvshl}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, |v|-1-s, 0), 0^s)$ for $|v| > s > 0$; $\texttt{bvshl}(v \in B, 0) = v$; $\texttt{bvshl}(v, s \geq |v|) = 0^{|v|}$
7. $\texttt{bvlshr}(v \in \mathcal{B}, s \in N) = \texttt{concat}(0^s, \texttt{extract}(v, |v|-1, s))$ for $|v| > s > 0$; $\texttt{bvlshr}(v \in B, 0) = v$; $\texttt{bvlshr}(v, s \geq |v|) = 0^{|v|}$
8. $\texttt{bvashr}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{repeat}(v^{[|v|-1]}, s), \texttt{extract}(v, |v|-1, s))$ for $|v| > s > 0$; $\texttt{bvashr}(v \in B, 0) = v$; $\texttt{bvashr}(v, s \geq |v|) = \texttt{repeat}(v^{[|v|-1]}, s)$
9. $\texttt{rotate\_left}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, (|v|-1-s)\%|v|, 0), \texttt{extract}(v, |v|-1, (|v|-s)\%|v|))$ for $s : s\%|v| \neq 0$; $\texttt{rotate\_left}(v, s) = v$ for $s : s\%|v| = 0$
10. $\texttt{rotate\_right}(v \in \mathcal{B}, s \in N) = \texttt{concat}(\texttt{extract}(v, (s-1)\%|v|, 0), \texttt{extract}(v, |v|-1, s\%|v|))$ for $s : s\%|v| \neq 0$; $\texttt{rotate\_right}(v, 0) = v$ for $s : s\%|v| = 0$

Figure 1: Bit-Propagating Bit-Vector Operations in the SMT-LIB 2.0 Language (% stands for the modulo operation)

`concat`. Note also that all the bit-propagating operations are equally applicable to bit-vector variables and constants.

3 Handling Bit-Propagating Operations

In this section we describe our algorithms for handling bit-propagating operations. Subsection 3.1 introduces the Bit-Propagating Normal Form (BPNF), while subsection 3.2 shows how to take advantage of BPNFs to boost the performance of the SMT solver.

3.1 Bit-Propagating Normal Form

We start with a definition of a segment, where a variable is *bit-propagating* if it was created as a result of applying a bit-propagating operation.

Definition 4 (Segment)**.** *A segment is either: (1) a bit-range of a non-bit-propagating variable, or (2) a constant. The set of all segments is denoted by* $\mathcal{S}$*.*

Bit-propagating variables can be expressed in terms of sequences of segments. Consider the example presented in Section 1. We would have: $v_2 = \left[v_1^{[2:0]}, 0^1\right]$, $v_3 = v_5 = \left[v_1^{[2:1]}\right]$ and

$v_4 = \left[v_1^{[3:0]}, v_1^{[3:0]}\right]$.

To define the standard form we need to make sure that adjacent segments are *merged*. For example, consider the variable v created by the following operation $v = \texttt{concat}(v^{[3:2]}, v^{[1:0]})$. The variable v could be expressed as any one of the following sequences of segments: $r_1 = \left[v^{[3:2]}, v^{[1:0]}\right]$, or $r_2 = \left[v^{[3:3]}, v^{[2:0]}\right]$, or $r_3 = \left[v^{[3:0]}\right]$. The last representation is the one that is desirable as the normal form. We formalize merge-related notions.

Definition 5 (Mergeable and Non-Mergeable Segments)**.** *Let $s_2, s_1 \in \mathcal{S}$ be two segments. The segments s_2 and s_1 (provided in that particular order) are* mergeable *iff one of the following conditions holds, otherwise they are* non-mergeable*:*

1. *Both s_2 and s_1 are constants, that is $s_2, s_1 \in \mathcal{C}$*
2. *Both s_2 and s_1 are bit-ranges, such that $s_2 = v^{[k:j+1]}$ and $s_1 = v^{[j:i]}$*

Definition 6 (Merge)**.** *Let $s_2, s_1 \in \mathcal{S}$ be two segments. The* merge *operation $\mathbb{M}(s_2, s_1)$ returns a sequence of one or two segments as follows:*

1. *If s_2 and s_1 are non-mergeable, $\mathbb{M}(s_2, s_1) = [s_2, s_1]$*
2. *If s_2 and s_1 are mergeable and are constants, $\mathbb{M}(s_2, s_1) = [\texttt{concat}(s_2, s_1)]$*
3. *If $s_2 = v^{[k:j+1]}$ and $s_1 = v^{[j:i]}$ are mergeable and are bit-ranges, $\mathbb{M}(s_2, s_1) = \left[v^{[k:i]}\right]$*

In our latest example (provided just before Def. 5), merging the two segments of r_1 and merging the two segments of r_2 results precisely in r_3 for both cases. We are now ready to introduce the Bit-Propagating Normal Form.

Definition 7 (Bit-Propagating Normal Form (BPNF))**.** *Given a bit-vector or a constant $t \in \mathcal{B} \cup \mathcal{C}$, the* bit-propagating normal form (BPNF) *$\Phi(t) = \left[\phi^t_{|\Phi(t)|-1} \in \mathcal{S}, \ldots, \phi^t_1 \in \mathcal{S}, \phi^t_0 \in \mathcal{S}\right]$ is a sequence of one or more segments, where for every $|\Phi(t)| - 2 > i \geq 0$, it holds that ϕ^t_{i+1} and ϕ^t_i are non-mergeable.*

In our example, we have $\Phi(v) = r_3$. Before presenting an algorithm for calculating the BPNF, we need some more definitions. We denote the number of bits in a segment $s \in S$ by $|s|$.

Definition 8 (Sub-segment)**.** *Let $s \in \mathcal{S}$ be a segment and i, j be numbers, such that $|s| > j, i \geq 0$ and $j \geq i$. Then the sub-segment $s^{[j:i]}$ is a new segment defined as follows:*

1. *If s is a constant $\left\{s_{|s|-1}, \ldots, s_1, s_0\right\}$, $s^{[j:i]} = \left\{s_j, \ldots, s_{i+1}, s_i\right\}$*
2. *If $s = v^{[k:l]}$ is a bit-range, $s^{[j:i]} = v^{[j+l:i+l]}$ (assuming $k \geq j + l$)*

It is not difficult to verify that a sub-segment is a segment. We will sometimes need to refer to the segment in $\Phi(v)$ of a bit of a given variable $v^{[i]}$ and the bit corresponding to $v^{[i]}$ in its segment.

Definition 9 (Segment Number, Segment Bit)**.** *Let $v^{[i]}$ be the i's bit of v. Let $s \geq 0$ be the largest number, such that $i \geq \sigma$, where $\sigma = \sum_{j=0}^{s-1} |\phi^v_j|$. Then, the* segment number *$sn(v^{[i]})$ and the* segment bit *$sb(v^{[i]})$ are defined as follows: $sn(v^{[i]}) = s$; $sb(v^{[i]}) = i - \sigma$.*

1. For a constant c: $\Phi(c) = [c]$.
2. For a non-bit-propagating variable v: $\Phi(v) = \left[v^{[|v|-1:0]}\right]$.
3. For a bit-propagating variable $v = \texttt{concat}(v_1, v_2)$: $\Phi(v) = \left[\phi^{v_2}_{|\Phi(v_2)|-1}, \ldots, \phi^{v_2}_2, \phi^{v_2}_1\right] \circ \bowtie(\phi^{v_2}_0, \phi^{v_1}_{|\Phi(v_1)|-1}) \circ \left[\phi^{v_1}_{|\Phi(v_1)|-2}, \ldots, \phi^{v_1}_1, \phi^{v_1}_0\right]$
4. For a bit-propagating variable $v = \texttt{extract}(u, m, l)$: $\Phi(v) = \left[\phi^{u[\mathrm{sb}(u^{[m]}),0]}_{\mathrm{sn}(u^{[m]})}, \phi^u_{\mathrm{sn}(u^{[m]})-1}, \ldots, \phi^u_{\mathrm{sn}(u^{[l]})+1}, \phi^{u\left[|\phi^u_{\mathrm{sn}(u^{[l]})}|-1, \mathrm{sb}(u^{[l]})\right]}_{\mathrm{sn}(u^{[l]})}\right]$
5. For a bit-propagating variable v created by neither `concat` nor `extract`, create $\Phi(v)$ by reducing the operation to applications of `concat` and `extract` as presented in Fig. 1.

Figure 2: Algorithm for calculating the Bit-Propagating Normal Form (BPNF) for a variable v. The operator $\circ$ stands for concatenation of sequences.

For example, given $v_2 = \left[v_1^{[2:0]}, 0^1\right]$, we have $\mathrm{sn}(v_2^{[0]}) = 0$; $\mathrm{sn}(v_2^{[1]}) = \mathrm{sn}(v_2^{[2]}) = \mathrm{sn}(v_2^{[3]}) = 1$; $\mathrm{sb}(v_2^{[0]}) = 0$; $\mathrm{sb}(v_2^{[1]}) = 0$; $\mathrm{sb}(v_2^{[2]}) = 1$; $\mathrm{sb}(v_2^{[3]}) = 2$.

In our approach, the SMT solver creates the BPNF for each new constant, input variable and internal variable representing the result of an operation. The algorithm for calculating the BPNF is provided in Fig. 2. Calculating the BPNF for constants, non-bit-propagating variables, and variables associated with the `extract` operation is straightforward. Finding the BPNF for `concat` requires concatenating the BPNFs of the two operands, where the BPNFs of the new neighbour pair are merged. Due of space limitations, we omit the proof that the algorithm in Fig. 2 returns a BPNF.

3.2 Implementation

In this section we show how to take advantage of BPNFs to speed-up the SMT solver.

Hashing BPNFs. To ensure that variables with the same BPNF are not created more than once, we maintain a hash table with all the current BPNFs and their corresponding variables. Whenever an operation is applied by the user, the solver creates a BPNF for a variable representing that operation (an actual variable is *not* created at this stage). If the BPNF appears in the hash table, its corresponding variable is returned to the user, otherwise a new variable is created and returned to the user, and the hash table is updated accordingly. The overhead of creating and maintaining the hash table is negligible in practice.

Using BPNFs for Translating to CNF. A major goal in introducing BPNFs is decreasing the number of CNF variables and clauses. This is achieved by never introducing any CNF variables or CNF clauses to represent bit-propagating variables and operations. Instead, to represent a bit-propagating variable v in CNF, we use the CNF variables that represent the non-bit-propagating variables that appear in v's BPNF.

User-Given Threshold on the Number of Segments. Maintaining too many segments in a BPNF might inflate the memory and lead to a performance degradation (at least in theory), because the algorithm for calculating BPNFs in Fig. 2 is linear in the number of segments. Hence we allow the user to impose a threshold, T, on the maximal number of segments permitted in a BPNF. If the number of segments for a variable v is greater than T, the variable will be considered to be a non-bit-propagating variable by the algorithm. Hence its BPNF will contain $v^{[|v|-1:0]}$, and the soundness of the SMT solution with respect to the corresponding bit-propagating operation will be ensured by bit-blasting that operation to CNF. We analyze the empirical impact of experimenting with different T values in Section 4.

Rewriting `assert`-based variable definitions. The SMT-LIB 2.0 language allows the user to build formulas in various ways. One of the common ways to create a new variable corresponding to a new operation is to use the `declare-fun` command to create a fake input variable and then to assert (using the `assert` command) that the new variable is, in fact, the result of an operation over existing variables. For example, the following sequence creates a new variable v that is defined to be $\texttt{repeat}(u, 2)$:

i. (`declare-fun` u () (_ `BitVec` 32)); ii. (`declare-fun` v () (_ `BitVec` 64)); iii. (`assert` (= v (`repeat` u 2)).

Such a way of creating variables is incompatible with our algorithm for calculating BPNFs, since our algorithm would consider variables bound to operations to be non-bit-propagating input variables. In our example, instead of figuring that $\Phi(v)$ is $\left[u^{[31:0]}, u^{[31:0]}\right]$, the algorithm would consider v to be a non-bit-propagating input variable with $\Phi(v) = \left[v^{[63:0]}\right]$. To overcome this problem, the preprocessor must identify such cases and rewrite them into a BPNF-friendly dag-oriented representation. In our example, rewriting the last `assert` command into the following form solves the problem: (`define-fun` v () (_ `BitVec` 64) (`repeat` u 2)).

The preprocessing algorithm for carrying out such rewriting is straightforward. Its complexity is linear in the size of the problem, and the overhead is still low. Moreover, such an algorithm can be seen as a particular case of term substitution [11], which in any event is implemented in modern solvers and is known to be useful for BV reasoning [11, 10].

Constant Propagation. Constant propagation is known to boost the performance of SMT solvers, and hence it is commonly used [11, 10]. It is essential to make sure that constant propagation is applied to take full advantage of BPNF-based algorithms over three shift operations, because there exist cases where the second operands of shift operations are not constants originally, but become constants after constant propagation. Recall that our algorithms consider shifts to be bit-propagating operations only when the second parameter is a constant.

4 Experimental Results

We carried out a number of experiments over SMT-LIB benchmark families from the QF_BV category to demonstrate the usefulness of our algorithms.

In the first experiment, we measured the proportion of bit-propagating operations in all the families (with the exception of the mcm family, whose benchmarks do not always conform to QF_BV syntax). The results are displayed in Table 1. One can see that for 37 families, the proportion of bit-propagating operations is at least 5%.

In our second (and main) experiment we ran Hazel over these 37 families (with the exception of all the sub-families of sage except app10 and app6, since they have a huge number of

benchmarks) with different T values. Recall from Section 3.2 that T is a user-given threshold value that limits the number of segments allowed in a BPNF. Note that our BPNF-based algorithms are disabled when $T = 0$. Recall also that Hazel is Intel's new eager BV solver. For the experiments we used machines running Intel® Xeon® processors with 3Ghz CPU frequency and having 32Gb of memory. The time-out for all the experiments was 600 sec. Table 2 shows the results for all the families where Hazel's cumulative run-time was more than 1 second for at least one configuration. Benchmarks where all the configurations timed-out are not considered in the table. The time-out value was added to the run-time when a memory-out occured.

One can see that our algorithms result in a performance boost in the case of 14 families. More specifically, for these families there exists at least one configuration of Hazel with $T \neq 0$ that outperforms the configuration with $T = 0$. The speed-up is at least 30% for eight of the families. The performance boost is especially significant for the top three families. The family spear/openldap v2.3.35 can only be solved when our algorithms are applied and T is high enough. The family pipe can only be solved with the configuration $T = 10$, while we observe a solid performance boost of over 2x for the family brummayerbiere for non-0 configurations. The choice between $T = 10$ and $T = 1000$ is family-specific, while an additional experiment has shown that increasing T from 1000 to 100000 does not change the performance.

Table 3 shows the number of word-level operations before bit-blasting the formula to CNF, as well as the number of CNF clauses and CNF variables for the configuration with $T = 0$. It also shows the ratio by which these numbers are reduced for configurations with $T = 10$ and $T = 1000$ as compared to the configuration with $T = 0$. The main conclusions to be drawn from the table are as follows. First, the number of word-level operations is only slightly reduced or not reduced at all, while the number of CNF clauses and variables is usually reduced considerably. This hints that the contribution to performance of BPNF-based translation to CNF is higher than that of BPNF hashing. Second, in most cases, the reduction in the number of CNF clauses and variables translates to a performance boost. However, this correlation is not absolute. Consider the family brutomesso/core, where our algorithms exhibited their worst performance. The number of clauses and variables was considerably reduced for that family. The reason for the lack of correlation in this case is apparently related to the sensitivity of SAT solver heuristics to the problem representation. We leave the study of this phenomenon to future reasearch.

Finally, to demonstrate that Hazel can compete with academic state-of-the-art SMT solvers, we ran Hazel against the latest versions of Boolector [7] (version 1.5.118), STP [12] (version 1373M), and Mathsat 5 [8] (with the configuration applied at the SMT'12 competition) over the eight families where use of our algorithms resulted in a performance boost of at least 30%. See Table 4 for the results. Hazel outperforms the academic solvers on all but one family. In our experiments, model generation was enabled for all the solvers. When model generation is disabled, the only significant change in run-time is for Boolector over the family uclid/catchconv, where the run-time is reduced to 702 seconds. Hazel is still much faster on this family.

5 Conclusion

Our goal was to improve bit-vector reasoning in modern SMT solvers. We identified a family of ten *bit-propagating* bit-vector operations in the SMT-LIB 2.0 language that fulfill the following property: each output bit is either a bit of one of the inputs or a constant (0 or 1). We demonstrated that bit-propagating operations are encountered frequently in SMT-LIB benchmarks. We proposed dedicated algorithms for handling such operations during SMT preprocessing and confirmed their empirical usefulness over a variety of SMT-LIB benchmark families.

Table 1: The number of benchmarks and the proportion of bit-propagating operations are provided per each SMT-LIB family in the QF_BV category. The families are sorted, in descending order, according to the proportion of bit-propagating operations. The sub-families of asp are not shown since the proportion is zero for all asp benchmarks.

Family	#	Proportion	Family	#	Proportion
uclid/tcas	2	0.61	calypto	23	0.56
sage/app11	611	0.5	bruttomesso/core	672	0.49
bench_ab	285	0.44	sage/app10	51	0.42
bruttomesso/lfsr	240	0.38	brummayerbiere2	65	0.37
uum	8	0.36	crafted	21	0.35
check	5	0.35	sage/app6	245	0.34
sage/app12	5784	0.29	wienand-cav2008/Booth	6	0.25
pipe	1	0.25	stp_samples	426	0.24
sage/app2	1417	0.22	check2	6	0.21
sage/app7	8663	0.19	brummayerbiere	52	0.17
sage/app5	1103	0.16	sage/app9	3301	0.16
sage/app8	2756	0.16	sage/app1	2676	0.14
spear/openldap_v2.3.35	8	0.14	galois	4	0.14
bruttomesso/simple_processor	64	0.13	uclid_contrib_smtcomp09	7	0.13
spear/inn_v2.4.3	219	0.08	wienand-cav2008/Commute	6	0.08
spear/wget_v1.10.2	42	0.08	spear/samba_v3.0.24	1386	0.08
wienand-cav2008/Distrib	6	0.07	uclid/catchconv	414	0.07
spear/xinetd_v2.3.14	2	0.06	stp	1	0.05
brummayerbiere3	79	0.05	spear/zebra_v0.95a	9	0.048
rubik	7	0.04	spear/cvs_v1.11.22	29	0.03
brummayerbiere4	10	0.02	dwp_formulas	332	0.01
asp (23 sub-families)	501	0	gulwani-pldi08	6	0
tacas07	5	0	VS3	11	0

Table 2: The impact of BPNF-based algorithms. We show the run-time of Hazel in seconds corresponding to 3 different T values (0, 10, 1000), the speedups of configurations with $T \neq 0$ over configuration with $T = 0$, and the number of solved instances corresponding to the 3 different T values. The results are sorted by the maximal speed-up over the configuration with $T = 0$. Best run-times are highlighted.

	Run-time in Seconds			Time Ratio		Solved Instances		
Family	Hzl_0	Hzl_10	Hzl_10^3	10/0	10^3/0	Hzl_0	Hzl_10	Hzl_10^3
spear/openldap_v2.3.35	1800	600	**19**	3.000	96.774	5	7	8
pipe	600	**155**	600	3.872	1.000	0	1	0
brummayerbiere	1649	**709**	711	2.326	2.321	40	41	41
wienand-cav2008/Booth	43	**26**	**26**	1.643	1.626	2	2	2
uum	18	**12**	**12**	1.535	1.534	2	2	2
brutomesso/simple_processor	374	**266**	**266**	1.404	1.405	64	64	64
uclid_contrib_smtcomp09	226	200	**169**	1.130	1.340	7	7	7
uclid/catchconv	9	8	**7**	1.115	1.312	414	414	414
brummayerbiere3	2921	**2600**	2885	1.123	1.012	42	42	42
brutomesso/lfsr	8039	**7435**	7439	1.081	1.081	230	227	227
spear/samba_v3.0.24	3516	**3359**	3433	1.047	1.024	1386	1386	1386
spear/inn_v2.4.3	624	**607**	708	1.027	0.880	219	219	219
stp	11	**10**	11	1.023	1.000	1	1	1
spear/wget_v1.10.2	308	319	**306**	0.966	1.006	42	42	42
brummayerbiere2	**719**	801	799	0.899	0.901	32	33	33
calypto	**213**	254	253	0.838	0.843	11	11	11
brutomesso/core	**19355**	24307	24307	0.796	0.796	933	925	925

6 Acknowledgments

The author would like to thank Paul Inbar for editing the paper and the anonymous reviewers whose valuable comments helped the author improve it.

References

[1] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *J. Autom. Reasoning*, 50(3):243–277, 2013.

Table 3: Hazel Statistics. The overall number of word-level operations, CNF clauses, and CNF variables are displayed for the configuration with $T = 0$ ('M' stands for millions). The reduction ratio of these parameters to the corresponding parameter for the configuration with $T = 0$ is displayed for the configurations with $T = 10$ and $T = 1000$. The families are sorted as in Table 2.

	0: Data Summary			**10: Reduction Ratio**			**1000: Reduction Ratio**		
Family	**Ops**	**Clss**	**Vars**	**Ops**	**Clss**	**Vars**	**Ops**	**Clss**	**Vars**
spear/openldap_v2.3.35	12469	3.86M	0.93M	1.0236	1.0986	1.2593	1.0236	1.1012	1.2661
pipe	1048	0.19M	92478	1.0029	2.7350	2.9764	1.0640	3.2986	3.6959
brummayerbiere	35351	14.5M	6.86M	1.0012	1.2373	1.2994	1.0012	1.2381	1.3004
wienand-cav2008/Booth	542	13471	3393	1	1.5158	1.3218	1	1.5199	1.3280
uum	1076	12605	5932	1	1.3812	1.3873	1	1.3812	1.3873
brutomesso/simple_processor	41736	1.66M	0.58M	1	1.3726	1.6326	1	1.3726	1.6326
uclid_contrib_smtcomp09	46156	1.84M	0.61M	1.0057	1.0584	1.0906	1.0068	1.0720	1.1125
uclid/catchconv	5.03M	63.5M	34.8M	1.0002	1.0711	1.0619	1.0002	1.0748	1.0631
brummayerbiere3	24289	2.46M	0.83M	1	1.0144	1.0137	1	1.0169	1.0175
brutomesso/lfsr	0.8M	71.9M	28.7M	1	1.2367	1.3153	1.0038	1.2399	1.3188
spear/samba_v3.0.24	9.4M	993M	338M	1	1.0322	1.0480	1	1.0380	1.0568
spear/inn_v2.4.3	0.11M	216M	46.4M	1.0048	1.0024	1.0055	1.0067	1.0040	1.0071
stp	0.32M	5.2M	3.39M	1	1.0025	1.0021	1	1.0025	1.0021
spear/wget_v1.10.2	15364	85.8M	19.6M	1	1.0006	1.0013	1.0017	1.0053	1.0019
brummayerbiere2	62564	71.9M	12.8M	1	1.3639	1.0038	1	1.3639	1.0038
calypto	11786	0.74M	0.29M	1.0448	1.3839	1.4321	1.0448	1.3963	1.4375
brutomesso/core	6.49M	106M	40.9M	1.0111	1.3442	1.4920	1.0370	1.3788	1.5262

Table 4: Comparing Hazel to Boolector, Mathsat, and STP. Best run-times are highlighted.

	Run-time in Seconds					Solved Instances				
Family	Btr	Mst	STP	Hzl_10	Hzl_10^3	Btr	Mst	STP	Hzl_10	Hzl_10^3
spear/openldap_v2.3.35	1924	1200	1204	600	**19**	5	6	6	7	8
pipe	325	600	600	**155**	600	1	0	0	1	0
brummayerbiere	907	4994	736	**709**	711	41	36	40	41	41
wienand-cav2008/Booth	21	**19**	45	26	26	2	2	2	2	2
uum	16	29	15	**12**	**12**	2	2	2	2	2
brutomesso/simple_processor	1908	22488	5156	**266**	**266**	64	29	59	64	64
uclid_contrib_smtcomp09	783	268	770	200	**169**	7	7	7	7	7
uclid/catchconv	9015	201	19	8	**7**	414	414	414	414	414

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. QF_BV Logic. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2, 2010.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[5] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC*, pages 522–527, 1998.

[6] Nikolaj Bjørner and Mark C. Pichora. Deciding Fixed and Non-fixed Size Bit-vectors. In *TACAS*, pages 376–392, 1998.

[7] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*, pages 174–177, 2009.

[8] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, pages 93–107, 2013.

[9] David Cyrluk, M. Oliver Möller, and Harald Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *CAV*, pages 60–71, 1997.

[10] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. Dissertation, University of Trento, 2010.

[11] Vijay Ganesh, Sergey Berezin, and David L. Dill. A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, Computer Science Department, Stanford University, April 2005.

[12] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, pages 519–531, 2007.